**北京邮电大学软件学院**

**2017-2018学年第1学期实验报告**

**课程名称： 操作系统**

**实验名称： 调度策略**

**实验完成人：**

**姓名：\_\_田宇\_\_学号：\_\_\_2016212011\_\_\_成绩：\_\_\_\_\_\_\_\_**

**指导教师：**\_\_\_\_\_\_\_**\_\_陈晋鹏\_\_**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**日 期： 2018 年 5 月 30 日**

1. **实验目的**

1. 理解 Linux 管理进程所用到的数据结构。

2. 理解 Linux 的进程调度算法的处理逻辑及其实现所用到的数据结构。

1. **实验内容**

1. 通过查阅参考书或者上网找资料，熟悉/usr/src/linux（注意：这里最后一级目录名可能是个含有具体内核版本号和“linux”字符串的名字）下各子目录的内容，即所含 Linux源代码的情况。

2. 分析 Linux 进程调度有关函数的源代码，主要是 schedule()函数，并且要对它们引用的头文件等一并分析。

3. 实现 Linux 的进程调度算法及理解其实现所用的主要数据结构。

1. **实验环境**

Window10 1709

1. **实验过程描述**

**//进程调度源代码分析**

**Schedule函数分析：**

schedule(void)

{

int i,next,c;

struct task\_struct \*\* p; //任务结构数组指针

//检测alarm（进程的报警定时值），唤醒任何已得到信号的可中断任务

for(p = &LAST\_TASK ; p > &FIRST\_TASK ; --p) //从任务结构数组中从后向前依次遍历数组

if (\*p) {

// 如果设置过任务的定时值alarm，并且已经过期(alarm<jiffies),则在信号位图中置SIGALRM 信号，

// 即向任务发送SIGALARM 信号。然后清alarm。该信号的默认操作是终止进程。

// jiffies 是系统从开机开始算起的滴答数（10ms/滴答）。定义在sched.h 第139 行。

if ((\*p)->alarm && (\*p)->alarm < jiffies) {

(\*p)->signal |= (1<<(SIGALRM-1));

(\*p)->alarm = 0;

}

// 如果信号位图中除被阻塞的信号外还有其它信号，并且任务处于可中断状态，则置任务为就绪状态。

// 其中'~(\_BLOCKABLE & (\*p)->blocked)'用于忽略被阻塞的信号，但SIGKILL 和SIGSTOP 不能被阻塞。

if (((\*p)->signal & ~(\_BLOCKABLE & (\*p)->blocked)) &&

(\*p)->state==TASK\_INTERRUPTIBLE)

(\*p)->state=TASK\_RUNNING;

}

while (1) {

c = -1;

next = 0;

i = NR\_TASKS; //任务结构数组长度，在0.11版本内核中为64

p = &task[NR\_TASKS];//任务结构数组指针

while (--i) {

if (!\*--p)

continue;//跳过任务结构数组中条项为零的不做处理

if ((\*p)->state == TASK\_RUNNING && (\*p)->counter > c)

c = (\*p)->counter, next = i; //找到进程表（数据结构表）中counter值最大的进程

}

if (c) break;找到则退出循环，switch到该进程

for(p = &LAST\_TASK ; p > &FIRST\_TASK ; --p) //程序运行到这里则所有任务的时间片均用完

// 根据每个任务的优先权值，更新每一个任务的counter 值

// counter 值的计算方式为counter = counter /2 + priority。这里计算过程不考虑进程的状态。

if (\*p)

(\*p)->counter = ((\*p)->counter >> 1) +

(\*p)->priority;

}

//如果系统中没有任何其他任务可运行，则next值为0，此时switch到任务0，仅执行pause（）系统调用，并又会调用本函数

switch\_to(next);

}

**Sleep\_on函数分析：**

void sleep\_on(struct task\_struct \*\*p)

{

struct task\_struct \*tmp;

if (!p) //若指针无效，则退出

return;

if (current == &(init\_task.task)) //若当前任务是任务0则死机

panic("task[0] trying to sleep");

tmp = \*p; //让tmp指向已经在等待队列上的任务

\*p = current; //将睡眠队列头的等待指针指向当前任务

current->state = TASK\_UNINTERRUPTIBLE; //将当前任务置为不可中断的等待状态

schedule(); //重新调度

// 只有当这个等待任务被唤醒时，调度程序才又返回到这里，则表示进程已被明确地唤醒。

if (tmp)

tmp->state=0; // 若在其前还存在等待的任务，则也将其置为就绪状态（唤醒）。

}

在几个进程为等待同一资源而多次调用该函数时，程序就隐式地构筑出一个等待队列。

在插入等待队列后，sleep\_on()函数就会调用schedule()函数去执行别的进程。当进程被唤醒而重新

执行时就会执行后续的语句，把比它早进入等待队列的一个进程唤醒。

**Wake\_up函数分析:**

void wake\_up(struct task\_struct \*\*p)

{

if (p && \*p) {

(\*\*p).state=0; // 置为就绪（可运行）状态。

\*p=NULL;

}

}

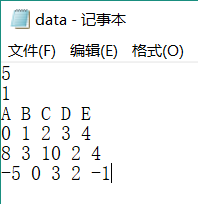
可知调用wake\_up(&p)时，此时的p指向最后一次调用sleep\_on()的task，即为task3,所以task3被唤醒，task3被唤醒，task3中的schedule()返回

执行后续代码，唤醒task2，于是task2中的schedule()返回，执行后续代码，唤醒task1，于是task1中的schedule()返回，task1中的tmp==NULL，

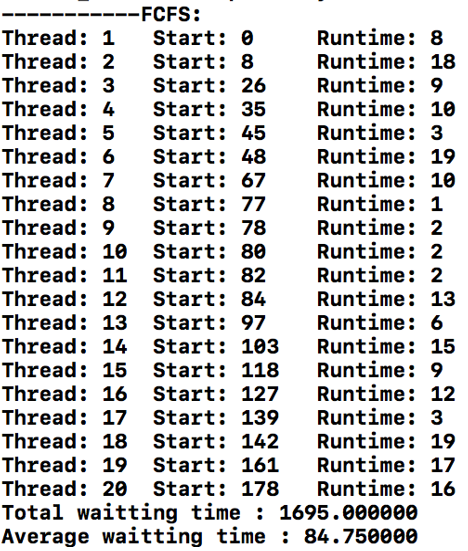
不执行后续代码，到此，因等待统一资源的三个进程都被唤醒；

1. **实验结果**

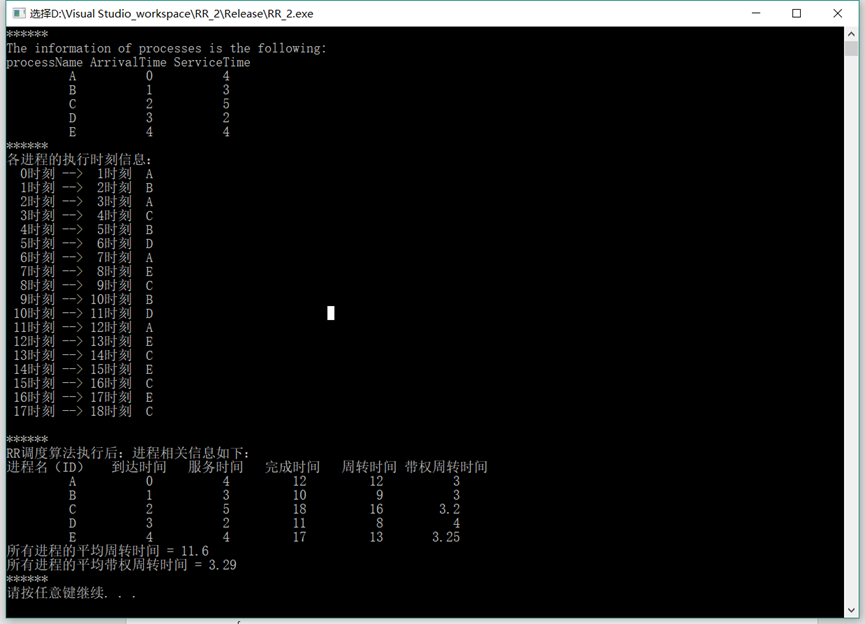
**1.SCHED\_OTHER截图：**



**2.SCHED\_FIFO截图：**



**3.SCHED\_RR截图：**



1. **附件**

**1. SCHED\_OTHER源代码:**

#include "stdafx.h"

#include <iostream>

#include <queue>

#include <iomanip>

#include <fstream>

using namespace std;

typedef struct

{

char name;

int ArrivalTime;

int Counter;

int NICE;

int Priority;

int FinishedTime;

}OTHER;

static queue<OTHER>OTHERqueue; //声明一个等待队列

static queue<OTHER>TEMPqueue; //声明一个用于排序的临时队列

static OTHER TEMPOTHER; //一个临时OTHER

static int q; //时间片

static int n; //进程个数

static OTHER OTHERarray[100]; //进程结构

void Input()

{

//文件读取模式

ifstream inData;

inData.open("data.txt");

inData>>n;

inData>>q;

for (int i=0;i<n;i++)

{

inData>>OTHERarray[i].name;

}

for (int i=0;i<n;i++)

{

inData>>OTHERarray[i].ArrivalTime;

}

for (int i=0;i<n;i++)

{

inData>>OTHERarray[i].Counter;

}

for (int i=0;i<n;i++)

{

inData>>OTHERarray[i].NICE;

}

cout<<"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"<<endl;

//输出用户所输入的信息

cout<<"The information of processes is the following:"<<endl;

cout<<setw(10)<<"processName"<<" ";

cout<<setw(10)<<"ArrivalTime"<<" ";

cout<<setw(10)<<"Counter"<<" ";

cout<<setw(10)<<"NICE"<<" "<<endl;

for (int i=0;i<n;i++)

{

cout<<setw(10)<<OTHERarray[i].name<<" ";

cout<<setw(10)<<OTHERarray[i].ArrivalTime<<" ";

cout<<setw(10)<<OTHERarray[i].Counter<<" ";

cout<<setw(10)<<OTHERarray[i].NICE<<" "<<endl;

}

cout<<"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"<<endl;

}

void OTHERAlgorithm()

{

char processMoment[100]; //存储每个时间片p对应的进程名称

OTHERqueue.push(OTHERarray[0]);

int processMomentPoint = 0;

int CurrentTime=0;

int tempTime; //声明此变量控制CurrentTime的累加时间，当前进程的服务时间小于时间片q的时候，起到重要作用

int i=1; //指向还未处理的进程的下标

int finalProcessNumber = 0; //执行OTHER算法后，进程的个数

int processTime[50];

//CurrentTime的初始化

if (OTHERarray[0].Counter>=q)

{

CurrentTime = q;

}

else

{

CurrentTime = OTHERarray[0].Counter;

}

while(!OTHERqueue.empty())

{

for (int j=i;j<n;j++) //使得满足进程的到达时间小于当前时间的进程都进入队列

{

if (OTHERarray[j].name!=NULL && CurrentTime >= OTHERarray[j].ArrivalTime)

{

OTHERarray[j].Priority=OTHERarray[j].Counter+20-OTHERarray[j].NICE;

OTHERqueue.push(OTHERarray[j]);

i++;

}

}

if (OTHERqueue.front().Counter<q)

{

tempTime = OTHERqueue.front().Counter;

}

else

{

tempTime = q;

}

OTHERqueue.front().Counter -= q; //进程每执行一次，就将其服务时间 -q

OTHERqueue.front().Priority = OTHERqueue.front().Counter+20-OTHERqueue.front().NICE;

//将队首进程的名称放入数组中

processMoment[processMomentPoint] = OTHERqueue.front().name;

processMomentPoint++;

processTime[finalProcessNumber] = tempTime;

finalProcessNumber++;

if (OTHERqueue.front().Counter <= 0) //把执行完的进程退出队列

{

//OTHERqueue.front().FinishedTime = CurrentTime;

OTHERqueue.pop(); //如果进程的服务时间小于等于，即该进程已经服务完了，将其退栈

}

else

{

//就绪队列里找一个（counter+20-nice）最大的线程运行

TEMPOTHER=OTHERqueue.front(); OTHERqueue.push(TEMPOTHER); OTHERqueue.pop();

while(!OTHERqueue.empty())

{

while(OTHERqueue.front().name!=TEMPOTHER.name)

{

if(OTHERqueue.front().Priority > TEMPOTHER.Priority)

{

//if(OTHERqueue.back().name!=TEMPOTHER.name)

// OTHERqueue.push(TEMPOTHER);

TEMPOTHER=OTHERqueue.front();

//OTHERqueue.pop();

}

OTHERqueue.push(OTHERqueue.front()); OTHERqueue.pop();

}

TEMPqueue.push(TEMPOTHER); OTHERqueue.pop();

TEMPOTHER=OTHERqueue.front(); OTHERqueue.push(TEMPOTHER); OTHERqueue.pop();

}

while(!TEMPqueue.empty())

{

OTHERqueue.push(TEMPqueue.front());TEMPqueue.pop();

}

}

CurrentTime += tempTime;

}

//进程输出处理 每个时间段对应的执行的进程

cout<<"各进程的执行时刻信息："<<endl;

cout<<" "<<"0时刻 --> "<<setw(2)<<processTime[0]<<"时刻";

processTime[finalProcessNumber]=0;

int time = processTime[0];

int count = 0;

for (int i=0;i<finalProcessNumber;i++)

{

count = 0;

cout<<setw(3)<<processMoment[i]<<setw(3)<<endl;

while(OTHERarray[count].name!=processMoment[i] && count<n)

{

count++;

}

OTHERarray[count].FinishedTime = time;

if (i<finalProcessNumber - 1)

{

cout<<setw(3)<<time<<"时刻"<<" --> "<<setw(2)<<time + processTime[i+1]<<"时刻"<<setw(3);

time += processTime[i+1];

}

}

cout<<endl;

}

void display()

{

cout<<"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"<<endl;

cout<<"OTHER调度算法执行后：进程相关信息如下："<<endl;

cout<<setw(10)<<"进程名（ID）"<<" ";

cout<<setw(10)<<"到达时间"<<" ";

cout<<setw(10)<<"服务时间"<<" ";

cout<<setw(10)<<"完成时间"<<" \n";

for (int i = 0;i<n;i++)

{

cout<<setw(10)<<OTHERarray[i].name<<" ";

cout<<setw(10)<<OTHERarray[i].ArrivalTime<<" ";

cout<<setw(10)<<OTHERarray[i].Counter<<" ";

cout<<setw(10)<<OTHERarray[i].FinishedTime<<" \n";

}

cout<<"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"<<endl;

}

/\*

int \_tmain(int argc, \_TCHAR\* argv[])

{

Input();

OTHERAlgorithm();

display();

return 0;

}

\*/

int main()

{

Input();

OTHERAlgorithm();

display();

return 0;

}

**2. SCHED\_FIFO源代码:**

#include "stdafx.h"

#include <iostream>

#include <time.h>

#include <iomanip>

#include <queue>

using namespace std;

typedef struct {

char name = '0';

int rt\_priority = 1;

int waitIO = 0;

int opreationComplete = 0;

}FIFO;

static queue<FIFO>FQueue[100];

static queue<FIFO>FWait[100];

static queue<FIFO>FSwap[100];

static int n;//\*\*

static int running;

static FIFO FArray[100];

void input() {

srand(NULL);

n = 5;

for (int i = 0; i < n; i++) {

FArray[i].name = (char)(65 + i);//ASCII A

FArray[i].rt\_priority = rand() / 99 + 1;//range 1-99

}

cout << setw(10) << "name" << setw(20) << "rt\_priority" << endl;

for (int i = 0; i < n; i++) {

cout << setw(10) << FArray[i].name << setw(20) << FArray[i].rt\_priority << endl;

}

}

void FIFOAlgorithm() {

FIFO swap;

for (int i = 0; i < n - 1; i++) {

for (int j = i + 1; j < n; j++) {

if (FArray[i].rt\_priority < FArray[i + j].rt\_priority){

swap = FArray[i];

FArray[i] = FArray[i + j];

FArray[i + j] = swap;

}

}

}//sort end

running = n;

while (!Fswap.empty()) Fswap.pop();

while (running != 0) {

while (FQueue.front().waitIO = 1) {

FWait.push(FQueue.front());

FQueue.pop();

}

cout << setw(10) << FQueue.front().name << "have been executed" << endl;

FQueue.pop();

running--;

while (FWait.front().waitIO = 0) {

FSwap.push(FWait.front());

FWait.pop();

}

while (!FQueue.empty()) {

FSwap.push(FQueue.front());

FQueue.pop();

}

while (!FSwap.empty()) {

FQueue.push(FSwap.front());

FSwap.pop();

}

}

}

int main() {

input();

FIFOAlgorithm();

system("pause");

return 0;

}

**3. SCHED\_RR源代码:**

// RR\_2.cpp: 定义控制台应用程序的入口点。

#include "stdafx.h"

#include <iostream>

#include <queue>

#include <iomanip>

#include <fstream>

using namespace std;

typedef struct

{

char name;

int ArrivalTime;

int ServiceTime;

int FinishedTime;

int WholeTime;

double WeightWholeTime;

}RR;

static queue<RR>RRqueue; //声明一个队列

static double AverageWT = 0, AverageWWT = 0;

static int q; //时间片

static int n; //进程个数

static RR RRarray[100]; //进程结构

void Input()

{

//文件读取模式

ifstream inData;

inData.open("C:\\Users\\qibai\\Desktop\\操作系统实验\\计算机操作系统实验代码（6个实验）\\时间片轮转RR进程调度算法\\data2.txt");

//data.txt表示q = 4的RR调度算法

//data2.txt表示q = 1的RR调度算法

inData >> n;

inData >> q;

for (int i = 0; i<n; i++)

{

inData >> RRarray[i].name;

}

for (int i = 0; i<n; i++)

{

inData >> RRarray[i].ArrivalTime;

}

for (int i = 0; i<n; i++)

{

inData >> RRarray[i].ServiceTime;

}

cout << "\*\*\*\*\*\*" << endl;

//输出用户所输入的信息

cout << "The information of processes is the following:" << endl;

cout << setw(10) << "processName" << " ";

cout << setw(10) << "ArrivalTime" << " ";

cout << setw(10) << "ServiceTime" << " " << endl;

for (int i = 0; i<n; i++)

{

cout << setw(10) << RRarray[i].name << " ";

cout << setw(10) << RRarray[i].ArrivalTime << " ";

cout << setw(10) << RRarray[i].ServiceTime << " " << endl;

}

cout << "\*\*\*\*\*\*" << endl;

}

void RRAlgorithm()

{

char processMoment[100]; //存储每个时间片p对应的进程名称

RRqueue.push(RRarray[0]);

int processMomentPoint = 0;

int CurrentTime = 0;

int tempTime; //声明此变量控制CurrentTime的累加时间，当前进程的服务时间小于时间片q的时候，起到重要作用

int i = 1; //指向还未处理的进程的下标

int finalProcessNumber = 0; //执行RR算法后，进程的个数

int processTime[50];

//CurrentTime的初始化

if (RRarray[0].ServiceTime >= q)

{

CurrentTime = q;

}

else

{

CurrentTime = RRarray[0].ServiceTime;

}

while (!RRqueue.empty())

{

for (int j = i; j<n; j++) //使得满足进程的到达时间小于当前时间的进程都进入队列

{

if (RRarray[j].name != NULL && CurrentTime >= RRarray[j].ArrivalTime)

{

RRqueue.push(RRarray[j]);

i++;

}

}

if (RRqueue.front().ServiceTime<q)

{

tempTime = RRqueue.front().ServiceTime;

}

else

{

tempTime = q;

}

RRqueue.front().ServiceTime -= q; //进程每执行一次，就将其服务时间 -q

//将队首进程的名称放入数组中

processMoment[processMomentPoint] = RRqueue.front().name;

processMomentPoint++;

processTime[finalProcessNumber] = tempTime;

finalProcessNumber++;

if (RRqueue.front().ServiceTime <= 0) //把执行完的进程退出队列

{

//RRqueue.front().FinishedTime = CurrentTime;

RRqueue.pop(); //如果进程的服务时间小于等于，即该进程已经服务完了，将其退栈

}

else

{

//将队首移到队尾

RRqueue.push(RRqueue.front());

RRqueue.pop();

}

CurrentTime += tempTime;

}

//进程输出处理 每个时间段对应的执行的进程

cout << "各进程的执行时刻信息：" << endl;

cout << " " << "0时刻 --> " << setw(2) << processTime[0] << "时刻";

processTime[finalProcessNumber] = 0;

int time = processTime[0];

int count = 0;

for (int i = 0; i<finalProcessNumber; i++)

{

count = 0;

cout << setw(3) << processMoment[i] << setw(3) << endl;

while (RRarray[count].name != processMoment[i] && count<n)

{

count++;

}

RRarray[count].FinishedTime = time;

if (i<finalProcessNumber - 1)

{

cout << setw(3) << time << "时刻" << " --> " << setw(2) << time + processTime[i + 1] << "时刻" << setw(3);

time += processTime[i + 1];

}

}

cout << endl;

//周转时间、带权周转时间、平均周转时间、带权平均周转时间的计算

//1. 周转时间 = 完成时间 - 到达时间

//2. 带权周转时间 = 周转时间/服务时间

for (int i = 0; i<n; i++)

{

RRarray[i].WholeTime = RRarray[i].FinishedTime - RRarray[i].ArrivalTime;

RRarray[i].WeightWholeTime = (double)RRarray[i].WholeTime / RRarray[i].ServiceTime;

}

double x = 0, y = 0;

for (int i = 0; i<n; i++)

{

x += RRarray[i].WholeTime;

y += RRarray[i].WeightWholeTime;

}

AverageWT = x / n;

AverageWWT = y / n;

}

void display()

{

cout << "\*\*\*\*\*\*" << endl;

cout << "RR调度算法执行后：进程相关信息如下：" << endl;

cout << setw(10) << "进程名（ID）" << " ";

cout << setw(10) << "到达时间" << " ";

cout << setw(10) << "服务时间" << " ";

cout << setw(10) << "完成时间" << " ";

cout << setw(10) << "周转时间" << " ";

cout << setw(10) << "带权周转时间" << endl;

for (int i = 0; i<n; i++)

{

cout << setw(10) << RRarray[i].name << " ";

cout << setw(10) << RRarray[i].ArrivalTime << " ";

cout << setw(10) << RRarray[i].ServiceTime << " ";

cout << setw(10) << RRarray[i].FinishedTime << " ";

cout << setw(10) << RRarray[i].WholeTime << " ";

cout << setw(10) << RRarray[i].WeightWholeTime << " " << endl;;

}

cout << "所有进程的平均周转时间 = " << AverageWT << endl;

cout << "所有进程的平均带权周转时间 = " << AverageWWT << endl;

cout << "\*\*\*\*\*\*" << endl;

}

int \_tmain(int argc, \_TCHAR\* argv[])

{

Input();

RRAlgorithm();

display();

system("pause");

return 0;

}